Date=1/12/2020
Lecture By=Manish Mahant
Subject ⇒ Combine reducers

IN PREVIOUS LECTURE **(QUICK RECAP) Date-30/11/2020**	**In Today's Lecture (Overview)**
Background Concepts State Management Immutability Store Dispatch Selectors Redux Application Data Flow#	combineReducers(reducers) Arguments# Returns# Notes# Example reducers/todos.js# App.js# Tips#

combineReducers(reducers)

As your app grows more complex, you'll want to split your reducing function into separate functions, each managing independent parts of the state.
The combineReducers helper function turns an object whose values are different reducing functions into a single reducing function you can pass to createStore.

The resulting reducer calls every child reducer, and gathers their results into a single state object. The state produced by combineReducers() namespaces the states of each reducer under their keys as passed to combineReducers()

Example:

```
rootReducer = combineReducers({potato: potatoReducer, tomato:
tomatoReducer})
// This would produce the following state object
{
  potato: {
    // ... potatoes, and other state managed by the potatoReducer ...
  },
  tomato: {
    // ... tomatoes, and other state managed by the tomatoReducer, maybe
some nice sauce? ...
  }
}
```

You can control state key names by using different keys for the reducers in the passed object. For example, you may call combineReducers({ todos: myTodosReducer, counter: myCounterReducer }) for the state shape to be { todos, counter }.
A popular convention is to name reducers after the state slices they manage, so you can use ES6 property shorthand notation: combineReducers({ counter, todos }). This is equivalent to writing combineReducers({ counter: counter, todos: todos }).

Arguments#

1. reducers (*Object*): An object whose values correspond to different reducing functions that need to be combined into one. See the notes below for some rules every passed reducer must follow.

Returns#

(*Function*): A reducer that invokes every reducer inside the reducers object, and constructs a state object with the same shape.

Notes#

This function is mildly opinionated and is skewed towards helping beginners avoid common pitfalls. This is why it attempts to enforce some rules that you don't have to follow if you write the root reducer manually.
Any reducer passed to combineReducers must satisfy these rules:

- For any action that is not recognized, it must return the state given to it as the first argument.
- It must never return undefined. It is too easy to do this by mistake via an early return statement, so combineReducers throws if you do that instead of letting the error manifest itself somewhere else.
- If the state given to it is undefined, it must return the initial state for this specific reducer. According to the previous rule, the initial state must not be undefined either. It is handy to specify it with ES6 optional arguments syntax, but you can also explicitly check the first argument for being undefined.

While combineReducers attempts to check that your reducers conform to some of these rules, you should remember them, and do your best to follow them. combineReducers will check your reducers by passing undefined to them; this is done even if you specify initial state to Redux.createStore(combineReducers(...), initialState). Therefore, you must ensure your reducers work properly when receiving undefined as state, even if you never intend for them to actually receive undefined in your own code.

Example

reducers/todos.js#

```
export default function todos(state = [], action) {
    switch (action.type) {
      case 'ADD_TODO':
        return state.concat([action.text])
      default:
        return state
    }
  }
```

App.js#

```
import { createStore } from 'redux'
import reducer from './reducers/index'

const store = createStore(reducer)
console.log(store.getState())
// {
//   counter: 0,
//   todos: []
// }

store.dispatch({
  type: 'ADD_TODO',
  text: 'Use Redux'
})
console.log(store.getState())
// {
//   counter: 0,
//   todos: [ 'Use Redux' ]
// }
```

Tips#

- This helper is just a convenience! You can write your own combineReducers that works differently, or even assemble the state object from the child reducers manually and write a root reducing function explicitly, like you would write any other function.
- You may call combineReducers at any level of the reducer hierarchy. It doesn't have to happen at the top. In fact you may use it again to split the child reducers that get too complicated into independent grandchildren, and so on.